UNIVERSITY OF EXETER
Q-STEP CENTRE

Q-Step
A step-change in quantitative social science skills
Funded by the Nuffield Foundation, ESRC and HEFCE

Data wrangling with Pandas and NumPy

Lewys Brace
l.brace@Exeter.ac.uk

Numerical Python (NumPy)

- NumPy is the most foundational package for numerical computing in Python.
- If you are going to work on data analysis or machine learning projects, then having a solid understanding of NumPy is nearly mandatory.
- Indeed, many other libraries, such as pandas and scikit-learn, use NumPy's array objects as the *lingua franca* for data exchange.
- One of the reasons as to why NumPy is so important for numerical computations is because it is designed for efficiency with large arrays of data. The reasons for this include:

    - It stores data internally in a continuous block of memory, independent of other in-built Python objects.

    - it performs complex computations on entire arrays without the need for for loops.

What you'll find in NumPy

- ndarray: an efficient multidimensional array providing fast array-orientated arithmetic operations and flexible *broadcasting* capabilities.
- Mathematical functions for fast operations on entire arrays of data without having to write loops.
- Tools for reading/writing array data to disk and working with memory-mapped files.
- Linear algebra, random number generation, and Fourier transform capabilities.
- A C API for connecting NumPy with libraries written in C, C++, and FORTRAN. This is why Python is the language of choice for wrapping legacy codebases.

The NumPy ndarray: A multi-dimensional array object

- The NumPy ndarray object is a fast and flexible container for large data sets in Python.
- NumPy arrays are a bit like Python lists, but still very different at the same time.
- Arrays enable you to store multiple items of the same data type. It is the facilities around the array object that makes NumPy so convenient for performing math and data manipulations.

Ndarray vs. lists

- By now, you are familiar with Python lists and how incredibly useful they are.
- So, you may be asking yourself:

  “I can store numbers and other objects in a python list itself and do all sorts of computations and manipulations through list comprehensions, for-loops etc. What do I need a NumPy array for?”
- There are very significant advantages of using NumPy arrays overs lists.

Creating a NumPy array

- To understand these advantages, lets create an array.
- One of the most common, of the many, ways to create a NumPy array is to create one from a list by passing it to the np.array() function.

In:
```
import numpy as np
list1 = [0, 1, 2, 3, 4]
arr = np.array(list1)

print(type(arr))
print(arr)
```

Out:
```
In [1]: runfile('C:/Users,
wdir='C:/Users/Lew_laptop,
<type 'numpy.ndarray'>
[0 1 2 3 4]
```

Differences between lists and ndarrays

- The key difference between an array and a list is, arrays are designed to handle vectorised operations while a python list is not.
- That means, if you apply a function it is performed on every item in the array, rather than on the whole array object.

- Let's suppose you want to add the number 2 to every item in the list. The intuitive way to do it is something like this:

In:
```
import numpy as np
list1 = [0, 1, 2, 3, 4]
list1 = list1+2
```

Out:
```
  File "C:/Users/Lew_laptop/.spyder-py3/temp.py", line 9, i
    list1 = list1+2

TypeError: can only concatenate list (not "int") to list
```

- That was not possible with a list, but you can do that on an array:

In:
```
import numpy as np
list1 = [0, 1, 2, 3, 4]
arr = np.array(list1)
print(arr)
arr = arr+2
print(arr)
```

Out:
```
In [7]: runfile('C:/Users
Lew_laptop/.spyder-py3')
[0 1 2 3 4]
[2 3 4 5 6]
```

- Another characteristic is that, once a Numpy array is created, you cannot increase its size. To do so, you will have to create a new array. But such a behaviour of extending the size is natural in a list.
- That said, there are so many more advantages.

Create a 2d array from a list of list

- You can pass a list of lists to create a matrix-like a 2d array.

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr2=np.array(list2)
print(arr2)
```

Out:
```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

The dtype argument

- You can specify the datatype by setting the dtype argument.
- Some of the most commonly used numpy dtypes are: float, int, bool, str, and object.

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print(arr3)
```

Out:

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

The astype argument

- You can also convert it to a different datatype using the astype method.

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print(arr3)
arr3_s = arr3.astype('int').astype('str')
print(arr3_s)
```

Out:

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
[['0' '1' '2']
 ['3' '4' '5']
 ['6' '7' '8']]
```

- Remember that, unlike list, all items in an array have to be of the same type.

dtype='object'

- However, if you are uncertain about what datatype your array will hold, or if you want to hold characters and numbers in the same array, you can set the dtype as 'object'.

```
In: arr_obj = np.array([1, 'a'], dtype='object')
    print(arr_obj)
```

```
Out: ['1' 'a']
```

The tolist() function

- You can always convert an array into a list using the tolist() command.

In:
```
arr_list = arr_obj.tolist()
print(arr_list)
```

Out:
```
[1, 'a']
```

Inspecting a NumPy array

- There are a range of functions built into NumPy that allow you to inspect different aspects of an array:

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print('Shape:', arr3.shape)
print('Data type:', arr3.dtype)
print('Size:', arr3.size)
print('Num dimensions:', arr3.ndim)
```

Out:
```
Shape: (3, 3)
Data type: float64
Size: 9
Num dimensions: 2
```

Extracting specific items from an array

- You can extract portions of the array using index, much like when you're working with lists.
- Unlike lists, however, arrays can optionally accept as many parameters in the square brackets as there is number of dimensions

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print("whole:", arr3)
print("Part:", arr3[:2, :2])
```

Out:
```
whole: [[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
Part: [[0. 1.]
 [3. 4.]]
```

Boolean indexing

- A boolean index array is of the same shape as the array-to-be-filtered and it contains only True and False values. The values corresponding to True positions are retained in the output.

```
In: import numpy as np
    list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
    arr3=np.array(list2, dtype='float')
    boo = arr3>2
    print(boo)
```

```
Out: [[False False False]
      [ True  True  True]
      [ True  True  True]]
```

Reversing rows and the whole array

- To reverse only the row positions:

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print('Original arrary:', arr3)
arr3_rr = arr3[::-1, ]
print('Rows reversed', arr3_rr)
```

Out:
```
Original arrary: [[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
Rows reversed [[6. 7. 8.]
 [3. 4. 5.]
 [0. 1. 2.]]
```

- To reverse both row and column positions:

In:
```
arr3_crr = arr3[::-1, ::-1]
print('Rows and columns reversed:', arr3_crr)
```

Out:
```
Rows and columns reversed: [[8. 7. 6.]
 [5. 4. 3.]
 [2. 1. 0.]]
```

Missing values in arrays

- Missing values can be represented using `np.nan` object. Similarly, infinite can be represented using `np.inf` :

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
arr3[1, 1] = np.nan
arr3[1, 2] = np.inf
print(arr3)
```

Out:
```
[[ 0.  1.  2.]
 [ 3. nan inf]
 [ 6.  7.  8.]]
```

Computing minimum and maximum values

- The ndarray has the following methods form computing the minimum and maximum values for the whole array:

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print('Min:', arr3.min())
print('Max:', arr3.max())
```

Out:
```
Min: 0.0
Max: 8.0
```

- If you want to compute these row or column rise:

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print('Col Min:', np.min(arr3, axis=0))
print('Col Max:', np.amin(arr3, axis=1))
```

Out:
```
Col Min: [0. 1. 2.]
Col Max: [0. 3. 6.]
```

Creating a new array from an existing one

- If you just assign a portion of an array to another array, the new array you just created actually refers to the parent array in memory.
- Therefore, any changes you make to this new array will be reflected in them original, parent, array as well.
- In order to avoid altering the original array, we use the `copy()` method.

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr=np.array(list2, dtype='float')
arr_b=arr
arr_b[:1, :1] = 100
print(arr_b)
print('_________')
arr=np.array(list2, dtype='float')
arr_b=arr.copy()
arr_b[:1, :1] = 101
print("arr", arr)
print("arr_b", arr_b)
```

Out:

```
[[100.   1.   2.]
 [  3.   4.   5.]
 [  6.   7.   8.]]
_________
arr [[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
arr_b [[101.   1.   2.]
 [  3.   4.   5.]
 [  6.   7.   8.]]
```

Creating sequences, repetitions and random numbers

- We are going to be using NumPy a lot in this course in order to create data sequences for our analysis.
- Indeed, the `np.arrage()` function is one that you are likely to use regularly in order to create customised number sequences as an ndarray.

In:

```
import numpy as np
# Lower limit is 0 be default
print(np.arange(5))
# 0 to 9
print(np.arange(0, 10))
# 0 to 9 with step of 2
print(np.arange(0, 10, 2))
# 10 to 1, decreasing order
print(np.arange(10, 0, -1))
```

Out:

```
[0 1 2 3 4]
[0 1 2 3 4 5 6 7 8 9]
[0 2 4 6 8]
[10  9  8  7  6  5  4  3  2  1]
```

- The np.arrange() allows you to set the starting and end positions.
- But if you are concerned with the number of items in the array, you will have to manually calculate the appropriate step values.
- For example, if you wanted an array of 10 numbers between 1 and 50, you could compute the step value.
- You could also is np.linspace() instead.

```
In:  import numpy as np
     print(np.linspace(start=1, stop=50, num=10, dtype=int))
Out: [ 1  6 11 17 22 28 33 39 44 50]
```

- Note: because I explicitly forced the dtype to be int, the numbers are not equally spaced because of the rounding.

- Similarly, there is also np.logspace() which rises in a logarithmic scale.
- In np.logspace(), the given start value is actually $base^{start}$ and ends with $base^{end}$, with a default based value of 10.

In:
```
import numpy as np
# Limit the number of digits after the decimal to 2
np.set_printoptions(precision=2)
# Start at 10^1 and end at 10^50
print(np.logspace(start=1, stop=50, num=10, base=10))
```

Out:
```
[1.00e+01 2.78e+06 7.74e+11 2.15e+17 5.99e+22 1.67e+28 4.64e+33 1.29e+39
 3.59e+44 1.00e+50]
```

- The np.zeros and np.ones functions lets you create arrays of desired shape where all the items are either 0's or 1's.

In:
```
import numpy as np
print(np.zeros([2,2]))
print(np.ones([2,2]))
```

Out
```
[[0. 0.]
 [0. 0.]]
[[1. 1.]
 [1. 1.]]
```

Creating repeating sequences

- The np.tile() will repeat a whole list or array *n* times.
- In comparison, np.repeat() repeats each item *n* times.

In:
```
import numpy as np
a=[1,2,3]
# Repeat whole of 'a' two times
print('Tile:   ', np.tile(a, 2))
# Repeat each element of 'a' two times
print('Repeat: ', np.repeat(a, 2))
```

Out:
```
Tile:   [1 2 3 1 2 3]
Repeat: [1 1 2 2 3 3]
```

Generating random numbers

- You will probably use the `random` module a lot.
- We'll be using it a lot in this course to generate random numbers and statistical distributions.

In:
```
import numpy as np
# Random numbers between [0,1) of shape 2,2
print(np.random.rand(2,2))
# Normal distribution with mean=0 and variance=1 of shape 2,2
print(np.random.randn(2,2))
```

Out:
```
[[0.71 0.38]
 [0.92 0.08]]
[[-0.07  1.16]
 [ 2.05 -0.29]]
```

In:

```
import numpy as np
# Random integers between [0, 10) of shape 2,2
print(np.random.randint(0, 10, size=[2,2]))
# One random number between [0,1)
print(np.random.random())
# Random numbers between [0,1) of shape 2,2
print(np.random.random(size=[2,2]))
# Pick 10 items from a given list, with equal probability
print(np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10))
# Pick 10 items from a given list with a predefined probability 'p'
print(np.random.choice(['a', 'e', 'i', 'o', 'u'], size=10, p=[0.3, .1, 0.1, 0.4, 0.1]))  # picks more o's
```

Out:

```
[[9 9]
 [1 3]]
0.7305288110727322
[[0.47 0.25]
 [0.75 0.65]]
['i' 'a' 'o' 'i' 'i' 'a' 'u' 'a' 'i' 'e']
['i' 'a' 'o' 'o' 'e' 'o' 'o' 'i' 'a' 'i']
```

- Every time you use any of the above functions, you will get a different set of random numbers.
- So, if you want to repeat the same set of random numbers every time, you need to set the *seed* or random state.
- The seed can be any number.
- Once np.random.RandomState is created, all the functions of the np.random module becomes available to the created randomstate object.

In:
```
import numpy as np
# Create the random state
rn = np.random.RandomState(100)
# Create random numbers between [0,1) of shape 2,2
print(rn.rand(2,2))
print('_____________')
# Set the random seed
np.random.seed(100)
# Create random numbers between [0,1) of shape 2,2
print(np.random.rand(2,2))
```

Out:
```
[[0.54 0.28]
 [0.42 0.84]]
_____________
[[0.54 0.28]
 [0.42 0.84]]
```

The np.unique() function

- The np.unique() method can be used to get the unique items. If you want the repetition counts of each item, set the return_counts parameter to True.

In:
```
import numpy as np
# Create random integers of size 10 between [0,10)
np.random.seed(100)
arr_rand = np.random.randint(0, 10, size=10)
print(arr_rand)
print('_____________________')
# Get the unique items and their counts
uniqs, counts = np.unique(arr_rand, return_counts=True)
print("Unique items : ", uniqs)
print("Counts       : ", counts)
```

Out:
```
[8 8 3 7 7 0 4 2 5 2]
_____________________
Unique items :  [0 2 3 4 5 7 8]
Counts       :  [1 2 1 1 1 2 2]
```

Pandas

- Pandas, like NumPy, is one of the most popular Python libraries for data analysis.
- It is a high-level abstraction over low-level NumPy, which is written in pure C.
- Pandas provides high-performance, easy-to-use data structures and data analysis tools.
- There are two main structures used by pandas; data frames and series.

Indices in a pandas series

- A pandas series is similar to a list, but differs in the fact that series associate a label with each element. This makes it look like a dictionary.
- If an index is not explicitly provided by the user, pandas creates a RangeIndex ranging from 0 to *N*-1.
- Each series object also has a data type.

In:
```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10])
print(new_series)
```

Out:
```
0     5
1     6
2     7
3     8
4     9
5    10
dtype: int64
```

- As you may suspect by this point, a series has attributes to extract ways to extract both all of the values in the series and individual elements by index.

```
In:  import pandas as pd
     new_series = pd.Series([5, 6, 7, 8, 9, 10])
     print(new_series.values)
     print('_____________')
     print(new_series[4])
```

```
Out:  [ 5  6  7  8  9 10]
      _____________
      9
```

- You can also provide an index manually.

```
In:  import pandas as pd
     new_series = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
     print(new_series.values)
     print('_____________')
     print(new_series['f'])
```

```
Out:  [ 5  6  7  8  9 10]
      _____________
      10
```

- It is easy to retrieve several elements of a series by their indexes or make group assignment.

In:

```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
print(new_series)
print('_____________')
new_series[['a', 'b', 'f']] = 0
print(new_series)
```

Out:

```
a     5
b     6
c     7
d     8
e     9
f    10
dtype: int64
_____________
a    0
b    0
c    7
d    8
e    9
f    0
dtype: int64
```

Filtering and maths operations

- Filtering and maths operations are easy with pandas as well.

In:
```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
new_series2 = new_series[new_series>0]
print(new_series2)
print('_____________')
new_series2[new_series2>0]*2
print(new_series2)
```

Out:
```
a     5
b     6
c     7
d     8
e     9
f    10
dtype: int64
_____________
a     5
b     6
c     7
d     8
e     9
f    10
dtype: int64
```

Pandas data frame

- Simplistically, a data frame is a table, with rows and columns.
- Each column in a data frame is a series object.
- Rows consist of elements inside series.

Creating a pandas data frame

- Pandas data frames can be constructed using Python dictionaries.

In:
```
import pandas as pd
df = pd.DataFrame({
    'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
    'population': [17.04, 143.5, 9.5, 45.5],
    'square': [2724902, 17125191, 207600, 603628]})
print(df)
```

Out:

	country	population	square
0	Kazakhstan	17.04	2724902
1	Russia	143.50	17125191
2	Belarus	9.50	207600
3	Ukraine	45.50	603628

- You can ascertain the type of a column with the `type()` function.

```
In:  print(type(df['country']))

Out: <class 'pandas.core.series.Series'>
```

- A pandas data frame object as two indices; a column index and row index.
- Again, if you do not provide one, pandas will create a RangeIndex from 0 to *N*-1.

In:
```
import pandas as pd
df = pd.DataFrame({
    'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
    'population': [17.04, 143.5, 9.5, 45.5],
    'square': [2724902, 17125191, 207600, 603628]})
print(df.columns)
print('__________')
print(df.index)
```

Out:
```
Index(['country', 'population', 'square'], dtype='object')

__________
RangeIndex(start=0, stop=4, step=1)
```

- There are numerous ways to provide row index explicitly.
- For example, you could provide an index when creating a data frame.

In:
```
import pandas as pd
df = pd.DataFrame({
    'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
    'population': [17.04, 143.5, 9.5, 45.5],
     'square': [2724902, 17125191, 207600, 603628]
}, index=['KZ', 'RU', 'BY', 'UA'])
print(df)
```

Out:

	country	population	square
KZ	Kazakhstan	17.04	2724902
RU	Russia	143.50	17125191
BY	Belarus	9.50	207600
UA	Ukraine	45.50	603628

- or do it on during runtime.
- I also name the index ‘country code’.

In:
```
import pandas as pd
df = pd.DataFrame({
    'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
    'population': [17.04, 143.5, 9.5, 45.5],
     'square': [2724902, 17125191, 207600, 603628]
})
print(df)
print('_____________')
df.index = ['KZ', 'RU', 'BY', 'UA']
df.index.name = 'Country Code'
print(df)
```

Out:

	country	population	square
0	Kazakhstan	17.04	2724902
1	Russia	143.50	17125191
2	Belarus	9.50	207600
3	Ukraine	45.50	603628

____

Country Code	country	population	square
KZ	Kazakhstan	17.04	2724902
RU	Russia	143.50	17125191
BY	Belarus	9.50	207600
UA	Ukraine	45.50	603628

- Row access using index can be performed in several ways.
- First, you could use .loc() and providing an index label.

```
In:  print(df.loc['KZ'])
```

```
Out: country       Kazakhstan
     population         17.04
     square           2724902
     Name: KZ, dtype: object
```

- Second, you could use .iloc() and providing an index number

```
In:  print(df.iloc[0])
```

```
Out: country       Kazakhstan
     population         17.04
     square           2724902
     Name: KZ, dtype: object
```

- Selection of particular rows and columns can be performed this way.

```
In: print(df.loc[['KZ', 'RU'], 'population'])
```

```
Out: Country Code
KZ     17.04
RU    143.50
Name: population, dtype: float64
```

- You can feed .loc() two arguments, index list and column list, slicing operation is supported as well:

```
In: print(df.loc['KZ':'BY', :])
```

```
Out:
              country  population    square
Country Code
KZ         Kazakhstan       17.04   2724902
RU             Russia      143.50  17125191
BY            Belarus        9.50    207600
```

Filtering

- Filtering is performed using so-called Boolean arrays.

```
print(df[df.population > 10][['country', 'square']])
```

Country Code	country	square
KZ	Kazakhstan	2724902
RU	Russia	17125191
UA	Ukraine	603628

Deleting columns

- You can delete a column using the drop() function.

In:
```
print(df)
```

Out:

Country Code	country	population	square
KZ	Kazakhstan	17.04	2724902
RU	Russia	143.50	17125191
BY	Belarus	9.50	207600
UA	Ukraine	45.50	603628

In:
```
df = df.drop(['population'], axis='columns')
print(df)
```

Out:

Country Code	country	square
KZ	Kazakhstan	2724902
RU	Russia	17125191
BY	Belarus	207600
UA	Ukraine	603628

Reading from and writing to a file

- pandas supports many popular file formats including CSV, XML, HTML, Excel, SQL, JSON, etc.
- Out of all of these, CSV is the file format that you will work with the most.
- You can read in the data from a CSV file using the `read_csv()` function.

```
df.to_csv('filename.csv')
```

- Similarly, you can write a data frame to a csv file with the `to_csv()` function.

```
df = pd.read_csv('filename.csv', sep=',')
```

- Pandas has the capacity to do much more than what we have covered here, such as grouping data and even data visualisation.
- However, as with NumPy, we don’t have enough time to cover every aspect of pandas here.

Any questions?